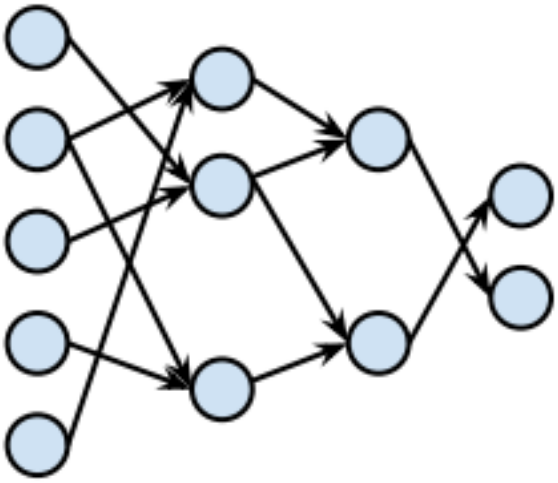


Bachelorscriptie

Reductie van Neurale Netwerken met GraphBLAS

Bart van der Lugt

Begeleider:
Prof. Dr. R.H. Bisseling

Universiteit Utrecht

Departement Wiskunde

13 juni 2017

Inhoudsopgave

1 Inleiding **2**

2 Graafalgoritmen **3**
2.1 Verbindingsmatrix 3
2.2 Semiring 4
2.3 Matrix-vectorvermenigvuldiging 6

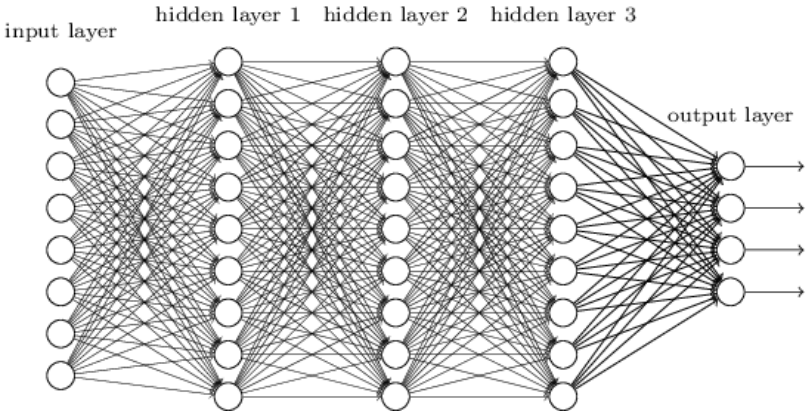
3 GraphBLAS API **7**
3.1 Objecten 7
3.2 Functies 8
3.3 Voordelen 9

4 Neurale Netwerken **9**
4.1 Classificatie 9
4.2 Leren met back-propagation 10
4.3 Toepassing: cijferherkenning 13
4.4 Uitbreiding: ijlheid 14

5 Implementatie **14**
5.1 Implementatie in GraphBlas 14
5.2 Pseudocode 15
5.3 Resultaten 16

6 Conclusie en Discussie **18**

A C++ Code Neuraal Netwerk **20**



Figuur 1: Basisschema neuraal netwerk

1 Inleiding

De afgelopen jaren is er sprake van een sterk groeiende interesse in *machine learning*. Hiermee wordt een deelgebied van de computerwetenschap aangeduid waarin men zich bezig houdt met computermodellen die kunnen *leren*. Het gaat hier echter om een vorm van leren waarvoor de computer niet expliciet geprogrammeerd is: in plaats daarvan worden algoritmes ontworpen om impliciete kennis te vergaren vanuit een omvangrijke dataset. Machine learning heeft veel raakvlakken met kunstmatige intelligentie.

Met name ontwikkelingen op het gebied van Neurale Netwerken hebben voor veel vooruitgang gezorgd wat betreft de nauwkeurigheid en de effectiviteit van machine learning. Dit is een datastructuur die zich namelijk bijzonder goed leent voor het leren aan de hand van voorbeelden. Een Neuraal Netwerk bestaat uit verschillende lagen met elementen die door middel van gewogen connecties verbonden zijn. De waarden van de gewichten bepalen de manier waarop een input (een element uit de dataset) wordt geclassificeerd. Over het algemeen wordt een laag opgeslagen als *vector* en een verzameling gewichten als *matrix.*

In traditionele Neurale Netwerken zijn aangrenzende lagen *all-to-all* verbonden: dat wil zeggen dat er vanuit elk element in de ene laag een connectie is met ieder element uit de andere laag. Dit heeft tot gevolg dat elke berekening in het model erg omvangrijk kan zijn, zelfs wanneer een heel aantal gewichten bijna nul of nul is. In de praktijk blijkt vaak dat dit voor veel connecties het geval is.

In deze scriptie wordt onderzoek gedaan naar een implementatie van een Neuraal Netwerk waarin geprofiteerd kan worden van *ijlheid* in de vectoren en matrices van het netwerk. Een matrix is ijl wanneer een groot aantal elementen de waarde nul heeft.

Hiervoor zal gebruik worden gemaakt van de GraphBLAS API. Dit is een verzameling objecten en functies gericht op het standaardiseren en optimaliseren van graafalgoritmen door gebruik te maken van vectoren en matrices. De hoop is dat deze primitieven zullen zorgen voor een makkelijke en overzichtelijke implementatie. Daarnaast lenen de gedefinieerde functies zich goed voor parallelle berekening en zijn ze vaak zeer efficiënt wat betreft rekentijd en opslag.

2 Graafalgoritmen

Dit hoofdstuk gaat in op het representeren van grafen en operaties op grafen in de taal van de lineaire algebra. Deze tak van de wiskunde is reeds uitgebreid onderzocht, zoals onder andere beschreven in [1]. Hier volgt eerst een korte definitie van grafen in de traditionele zin. Een graaf is intuïtief gedefinieerd als een verzameling punten en een verzameling kanten die de punten in de graaf onderling verbinden. Formeel ziet dit er als volgt uit:

Definitie 2.0.1. *Een graaf is een geordend paar $G = (V, E)$ waarin V een verzameling is genaamd de* puntenverzameling*, E een verzameling van paren elementen uit V genaamd de* kantenverzameling*, zodanig dat: $E \subseteq \{\{u, v\} : u, v \in V\}$.*

Wanneer een paar punten $(u, v) \in E$ zeggen we dat punt u met punt v verbonden is door middel van een kant. Een alternatieve benaming voor deze relatie is dat u en v buren zijn. In een ongerichte graaf geldt dat een verbinding bidirectioneel is. Dat betekent dat bij de beschouwing van een paar $(u, v) \in E$ de volgorde van u en v niet belangrijk is. In een gerichte graaf is de volgorde wel van belang: we spreken dan bij een verbinding $(u, v) \in E$ alleen van een kant tussen u en v, en per definitie niet van een kant tussen v en u. In dat geval vervangt men daarom de term 'kant' vaak door 'pijl'.

2.1 Verbindingsmatrix

Voor het representeren van een graaf kan gebruik gemaakt worden van een matrix. Men spreekt dan van de verbindingsmatrix behorende bij graaf G. Hierin zijn twee mogelijkheden: men kan gebruik maken van een *bogenmatrix* of een *incidentiematrix*. De bogenmatrix is zowel intuïtief als operationeel gezien de meest voor de hand liggende keuze: in deze scriptie komt daarom alleen de bogenmatrix aan de orde.

Definitie 2.1.1. *Gegeven een graaf $G(V, E)$ en de bogenmatrix A_G behorende bij G. Dan geldt: A_G is een matrix met dimensie $|V| \times |V|$ gegeven door*

$$A_G(i, j) = \begin{cases} 1 & \text{wanneer} \quad (v_i, v_j) \in E \quad , \quad v_i, v_j \in V \\ 0 & \text{anders} \end{cases}$$

Voor het gebruik van de bogenmatrix is het dus van belang dat ieder punt in V een unieke index heeft. Iedere kant in de graaf wordt gerepresenteerd in de matrix als een 1: de rijindex wordt gebruikt om het startpunt van de kant aan te geven, en de kolomindex voor het eindpunt. Indien graaf G ongericht is geldt dat $A_G(i, j) = 1 \Leftrightarrow A_G(j, i) = 1$. Dit volgt uit het feit dat de verbindingen in dat geval bidirectioneel zijn.

Tot slot is het ook mogelijk een numerieke waarde te associëren met een kant. In dat geval spreekt men vaak van het *gewicht* van de kant. Dit gewicht kan gebruikt worden in operaties die worden uitgevoerd op de graaf. In de matrix wordt op de plek die overeen komt met de kant in plaats van de waarde 1 dan vaak de waarde van het gewicht opgeslagen.

2.2 Semiring

Voor het uitvoeren van berekeningen met verbindingsmatrices is de algebraïsche definitie van een semiring erg belangrijk. Deze dient als generalisatie van het tweetal operaties dat toegepast kan worden op de elementen van de matrices.

Definitie 2.2.1. *Een semiring is een verzameling S uitgerust met twee binaire operatoren $\oplus$ en $\otimes$, zodanig dat wordt voldaan aan de volgende eisen:*
Additieve associativiteit $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
Additieve commutativiteit: $a \oplus b = b \oplus a$
Multiplicatieve associativiteit: $(a \otimes b) \otimes c = a \otimes (b \otimes c)$
Distributiviteit: $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$

Het simpelste voorbeeld van een semiring zijn de natuurlijke getallen $\mathbb{N}$ uitgerust met de standaard operatoren $+$ en $*$ voor optellen en vermenigvuldigen.

Met behulp van de operaties in de semiring kunnen twee matrices van hetzelfde formaat elementsgewijs met elkaar worden gecombineerd. Gegeven matrices $\mathbf{A}$, $\mathbf{B}$ en $\mathbf{C}$ gaat dit als volgt:
Elementsgewijs optellen:

$$\mathbf{A} \oplus \mathbf{B} = \mathbf{C}$$

is gegeven door $\mathbf{C}(i,j) = \mathbf{A}(i,j) \oplus \mathbf{B}(i,j)$.
Elementsgewijs vermenigvuldigen:

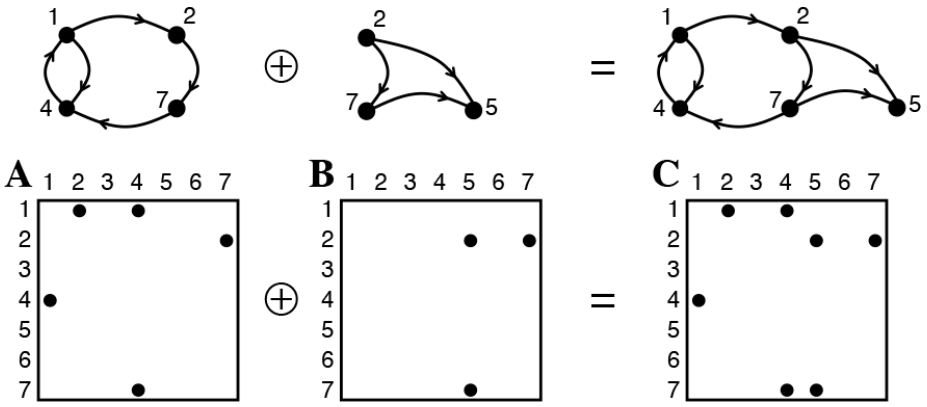
$$\mathbf{A} \otimes \mathbf{B} = \mathbf{C}$$

is gegeven door $\mathbf{C}(i,j) = \mathbf{A}(i,j) \otimes \mathbf{B}(i,j)$.

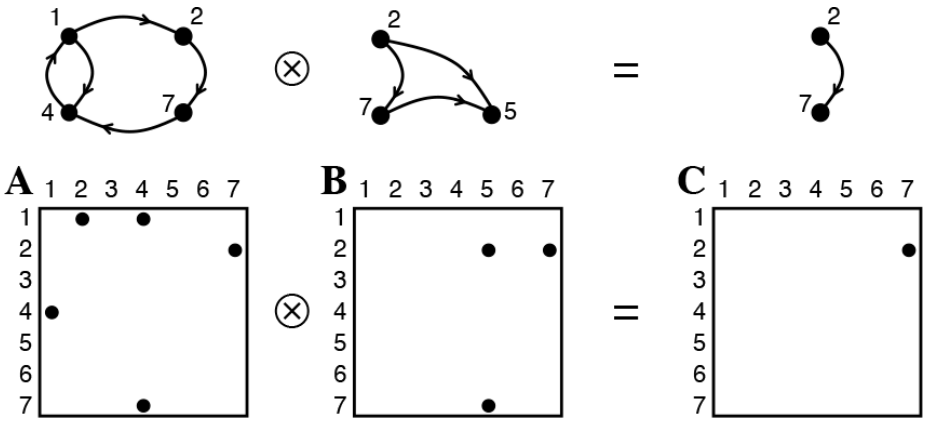
De eigenschappen van de semiring verzekeren ons ervan dat de volgorde van de matrices in de berekening niet van belang is.

De semiring die wordt gekozen definieert een unieke manier waarop een tweetal matrices met elkaar samenhangt. Er zijn graafalgoritmen waarvoor specifieke operaties wenselijk of nodig zijn. Voorbeelden van semiringen die kunnen worden gebruikt zijn:
$(S, \oplus, \otimes)$ = $\mathbb{N}$, max, $+$
$(S, \oplus, \otimes)$ = $\mathbb{N}$, min, max
$(S, \oplus, \otimes)$ = $\{0, 1\}$, $\vee$, $\wedge$
Wanneer men de standaard operatoren voor optellen en vermenigvuldigen gebruikt definieert het elementsgewijs optellen van twee bogenmatrices het combineren van de twee bijbehorende grafen tot één nieuwe graaf. Evenzo definieert het elementsgewijs vermenigvuldigen van twee bogenmatrices het snijden van de twee grafen. In figuren 2 en 3 is een illustratie te vinden van beide matrixberekeningen met de bijbehorende representatie in de vorm van een graaf.

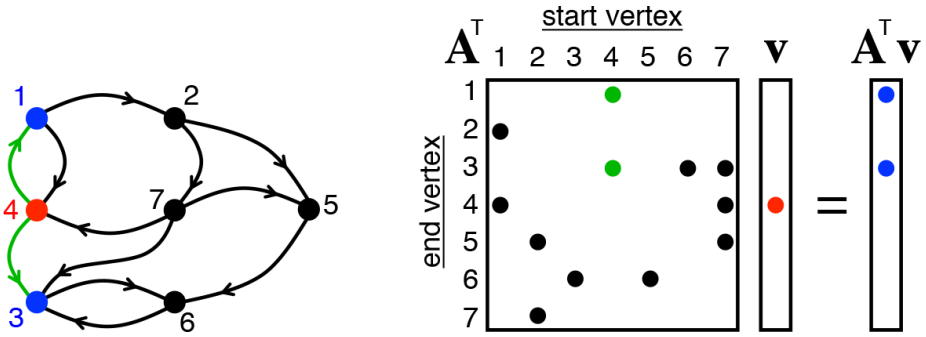
Bij figuur 2 moet worden opgemerkt dat kant (2,7) tweemaal in het product voorkomt. Indien men dit toelaat spreekt men van een multigraaf, waarin het mogelijk is dat er meerdere kanten zijn tussen een tweetal punten. In deze scriptie worden enkel *simpele grafen* beschouwd, waarin dat niet mogelijk is. Daarom moeten we van de productmatrix $\mathbf{C} = \mathbf{A} \oplus \mathbf{B}$ eisen dat alle waarden groter dan 1 worden veranderd in 1.



Figuur 2: Illustratie van het optellen van twee bogenmatrices met de representatie behorende bij de operatie in de vorm van een graaf, indien $\oplus = +$



Figuur 3: Illustratie van het vermenigvuldigen van twee bogenmatrices met de representatie behorende bij de operatie in de vorm van een graaf, indien $\otimes = *$



Figuur 4: Illustratie van het vinden van 'buren' gegeven één startpunt door het matrix-vector product met de bogenmatrix te nemen.

2.3 Matrix-vectorvermenigvuldiging

In de lineaire algebra is er nog een manier om een tweetal matrices met elkaar te verbinden, en dat is door middel van het matrixproduct. Het matrixproduct kan berekend worden van twee matrices $\mathbf{A}$ en $\mathbf{B}$ wanneer het aantal kolommen van matrix $\mathbf{A}$ gelijk is aan het aantal rijen van matrix $\mathbf{B}$. De definitie, uitgaande van de standaard operatoren voor optellen en vermenigvuldigen, is als volgt:

Definitie 2.3.1. *Gegeven matrix $\boldsymbol{A}$ met dimensie $n \times m$ en matrix $\boldsymbol{B}$ met dimensie $m \times p$, dan is het matrixproduct $\boldsymbol{AB} = \boldsymbol{C}$, met $\boldsymbol{C}$ de productmatrix met dimensie $n \times p$, gegeven door:*

$$\boldsymbol{C}(i,j) = \sum_{k=1}^{m} \boldsymbol{A}(i,k) * \boldsymbol{B}(k,j)$$

Wanneer men kiest voor een semiring met een ander tweetal operatoren geven we de algemene definitie van het matrixproduct als volgt weer:

$$\mathbf{C}(i,j) = \bigoplus_{k=1}^{m} \mathbf{A}(i,k) \otimes \mathbf{B}(k,j)$$

Met bovenstaande definitie is ook het vector-matrix product (waarin $\mathbf{A}$ van dimensie $1 \times l$) en het matrix-vector product (waarin $\mathbf{B}$ van dimensie $l \times 1$) gegeven. Dit matrix-vector product heeft een belangrijke toepassing voor graaf-algoritmen. De berekening kan, uitgerust met de standaard semiring, namelijk gebruikt worden om de buren te vinden van een punt in de graaf. Daarmee wordt bedoeld dat, gegeven een startpunt, alle punten worden gevraagd die verbonden zijn met dit startpunt. Hiervoor wordt het product genomen van de bogenmatrix en een vector welke één of meerdere startpunten binnen de graaf bevat. Het resultaat is een vector met daarin slechts de buren van deze startpunten. Een illustratie hiervan is gegeven in figuur 4. Hierin is de matrix $\mathbf{A}$ de bogenmatrix behorende bij de graaf, en bevat de vector $\mathbf{v}$ een enkel punt uit de matrix.

Het matrixproduct heeft ook een bijzondere toepassing indien de bogenmatrix wordt gebruikt om de gewichten van de kanten op te slaan. In dat geval bevatten de vector en de productvector vaak ook numerieke waarden in plaats

van slechts een 1 op de juiste plek. Deze waarde hoort dan bij een enkel punt in de matrix en wordt vaak de *activatie* van het betreffende punt genoemd. Iedere activatiewaarde in de productvector wordt berekend door de som te nemen van de activatiewaarden van de buren vermenigvuldigd met het gewicht van de verbindende kant. Deze toepassing van de operatie is ook bevat in de definitie van het matrixproduct zoals gegeven in Definitie 2.3.1.

3 GraphBLAS API

In het vorige hoofdstuk is het verband uitgelegd tussen operaties op grafen en berekeningen met matrices. Dit hoofdstuk beschouwt de definities en de specificaties van de implementatie van het GraphBLAS project: [2], [3]. Het doel van de GraphBLAS is het ontwikkelen van een collectie primitieve bouwstenen voor graafalgoritmen, gedefinieerd in de taal van lineaire algebra. Het project bouwt voort op de daarvoor gedefinieerde standaard van de NIST Sparse Basic Linear Algebra Subprograms (BLAS) [4]. De operaties die hierin werden beschreven hadden betrekking op ijle vectoren (niveau 1), vector matrix (niveau 2), en matrix matrix (niveau 3) en maakten enkel gebruik van de traditionele optelling en vermenigvuldiging. In de jaren daarna werd de BLAS uitgebreid met meer algemene definities en met de combinatorial BLAS [5] werd aangetoond dat deze aanpak leidt tot de mogelijkheid een breed scala graafalgoritmen te implementeren.

In het streven naar deze standaard zijn een aantal dingen van belang. Allereerst moet de verzameling basisfunctionaliteiten breed genoeg zijn voor de verschillende toepassingen van graafalgoritmen. Daarnaast is het belangrijk dat de operaties parallel kunnen worden uitgevoerd om op die manier de effectiviteit te vergroten. Ook de gekozen datatypes zijn van belang, gezien de noodzaak om zowel alle standaard types (boolean, float, double) te bevatten alsmede ondersteuning te bieden aan symmetrische matrices, tuples en gecomprimeerde ijle vectoren.

3.1 Objecten

GraphBLAS biedt ondersteuning aan de volgende primitieve objecten, welke gebruikt kunnen worden om gegevens in op te slaan en operaties mee uit te voeren.

Vector Een vector is een array van vooraf gespecificeerde grootte waarin iedere plaats gevuld is met één scalar. Veel programmeertalen bieden standaard al ondersteuning aan vectoren. Wanneer vectoren voor graafalgoritmen worden gebruikt bevatten vectoren vaak veel elementen met waarde 0: de vector is dan ijl. Vandaar dat er in de GraphBLAS voor gekozen is een structural zero te implementeren: elementen met waarde 0 worden niet daadwerkelijk opgeslagen in de array, maar de bijbehorende plaats blijft leeg. Op die manier wordt geheugen bespaard.

Matrix Een matrix van grootte $n \times m$ kan worden opgeslagen als een verzameling van n arrays van grootte m: de arrays stellen de rijen van de matrix voor. Hierin worden de waarden opgeslagen en wordt wederom gebruik gemaakt van een structural zero.

Compressed Sparse Matrix Omdat vectoren in graafalgoritmen vaak ijl zijn biedt GraphBLAS ook ondersteuning aan gecomprimeerde matrices. Hierin worden de waarden niet opgeslagen in een array, maar als een lijst met tuples. Een tuple is in dit geval een geordend drietal getallen: een rijindex, een kolomindex, en een waarde. Dit bespaart erg veel geheugen indien de matrix ijl is, aangezien er nu geen geheugenruimte wordt gereserveerd voor elementen met waarde 0. Ook kan de definitie van de gecomprimeerde matrix worden gebruikt voor vectoren: Bij een rijvector is dan de rijindex altijd 1, bij een kolomvector is de kolomindex altijd 1.

Semiring GraphBLAS biedt de mogelijkheid om een eigen semiring te definiëren. Dit is vooral handig indien de gebruiker de twee binaire operatoren waarmee de semiring is uitgerust zelf wil implementeren. Hierdoor is het gebruik van de GraphBLAS bijzonder veelzijdig.

3.2 Functies

Buildmatrix De functie buildmatrix kan gebruikt worden om een vector, matrix, of compressed matrix object te maken. De input voor deze methode is een serie tuples met rijindices, kolomindices, en waarden. Op deze manier kan bijvoorbeeld een graaf worden gerepresenteerd in de code aan de hand van de startpunten en eindpunten.

ExtracTuples geeft de volledige inhoud van een matrix of vector als een lijst met tuples. Een tuple bevat een rijindex, een kolomindex en een waarde.

MxM Deze functie implementeert het matrixproduct. De invoer zijn drie matrices: matrices $\mathbf{A}$ en $\mathbf{B}$ waarvan het product wordt genomen, en matrix $\mathbf{C}$ waarin het resultaat wordt opgeslagen. Deze moeten van het juiste formaat zijn. Ook is het mogelijk aan de MxM-functie een zelf gedefinieerde semiring mee te geven, waardoor de matrixberekening op een unieke manier kan worden uitgevoerd.

MxV en VxM Deze functies implementeren hetzelfde als de functie hierboven, alleen zijn deze gericht op en geoptimaliseerd voor de matrix-vector berekening. In het geval van MxV moet matrix $\mathbf{B}$ dus van dimensie $m \times 1$ zijn, en bij VxM moet matrix $\mathbf{A}$ van dimensie $1 \times m$ zijn.

Transpose Deze functie transponeert een matrix $\mathbf{A}$, en slaat de waarden op in matrix $\mathbf{A}^\top$. Transponeren betekent dat de rij- en kolomindices worden omgedraaid. Dit resulteert voor een matrix $\mathbf{A}$ van dimensie $n \times m$ in een getransponeerde matrix $\mathbf{A}^\top$ van dimensie $m \times n$.

EwiseMult, EwiseAdd Met deze functies kunnen twee matrices of vectoren van hetzelfde formaat worden gecombineerd. Het is hierin mogelijk om een zelf gedefinieerde semiring mee te geven aan de functie, zodat de manier waarop het product of de som van de invoeren wordt samengesteld uniek bepaald kan worden.

Apply Tot slot is het in de GraphBLAS mogelijk een unaire functie toe te passen op alle elementen in een vector of matrix. Deze kan door de gebruiker gedefinieerd worden, en heeft als toepassing dat de elementen kunnen worden geschaald of afgeschat.

3.3 Voordelen

Er zijn verschillende redenen waarom de GraphBLAS een uitzonderlijk goede keuze is als software om algoritmen te implementeren die werken met lineaire algebra. Allereerst definieert de API een brede, algemene standaard. Er is veel ruimte voor de gebruiker om functies te definiëren en op die manier op een unieke manier operaties op de matrices en vectoren uit te voeren. Hierdoor is de GraphBLAS geschikt voor veel verschillende en uiteenlopende algoritmen. Dit zou het onderzoek op het gebied van graafalgoritmen sterk vooruit helpen aangezien onderzoekers niet voor ieder nieuw project een nieuwe programmeertaal hoeven aan te leren, maar telkens op de hoogte kunnen blijven van de manier waarop algoritmen geïmplementeerd worden.

Daarnaast maakt de GraphBLAS op een zeer efficiënte manier gebruik van de ijlheid in vectoren en matrices. De mogelijkheid om gemakkelijk berekeningen uit te voeren met matrices die gecomprimeerd opgeslagen zijn als tuples is enorm voordelig. Het scheelt veel rekentijd en opslag indien kan worden geprofiteerd van deze eigenschap, maar een operatie die efficient kan worden toegepast op zowel een ijle als dichte matrix is complex. De GraphBLAS bevat deze operaties al van zichzelf, hetgeen het gebruik van de API aantrekkelijker maakt.

Tot slot leent de GraphBLAS API zich zeer goed voor het parallel uitvoeren van operaties op verschillende processoren. Er wordt namelijk voor gekozen de objecten en functies zo te implementeren dat alle operaties verplicht door de API moeten worden gedaan. Praktisch gezien leidt dit tot minder vrijheid bij de gebruiker. Echter, het grote voordeel hieraan is dat de berekening achter de schermen van de GraphBLAS API zo efficient mogelijk kan worden gedaan zonder dat de gebruiker zich hiervoor hoeft in te spannen tijdens het ontwikkelen van het algoritme.

4 Neurale Netwerken

4.1 Classificatie

Een neuraal netwerk is qua architectuur gebaseerd op het menselijk brein. De datastructuur is opgebouwd uit verschillende, geordende lagen met neuronen. De neuronen stellen punten in een netwerk voor, en zijn verbonden met de neuronen in voorgaande en opvolgende lagen door middel van gewogen kanten. Iedere neuron heeft een activatiewaarde, en de activatiewaarden van de neuronen in één laag worden opgeslagen in een vector. Een illustratie van een neuraal netwerk, gerepresenteerd als een graaf, is te zien in figuur 1. Om bovenstaande vast te leggen voeren we wat notatie in en werken we dit uit tot een definitie. In hoofdstuk 2 hebben we de relatie gezien tussen een graaf en de representatie daarvan in de taal van de lineaire algebra. De definities in dit hoofdstuk bouwen daarop voort.

Definitie 4.1.1. *Een neuraal netwerk N bestaat uit:*

- *een vector* $\mathbf{x}_0$ *welke de inputvector heet,*
- m *vectoren* $\mathbf{x}_1, .., \mathbf{x}_m$ *welke verborgen vectoren heten,*
- *een vector* $\mathbf{x}_{m+1}$ *welke de outputvector heet,*
- $m+1$ *matrices* $\mathbf{W}_1, .., \mathbf{W}_{m+1}$*, welke de gewichtenmatrices heten, waarvoor geldt:* $\mathbf{W}_i$ *van dimensies* $|\mathbf{x}_i| \times |\mathbf{x}_{i-1}|$ *voor* $1 \leq i \leq m+1$*.*

De inputvector wordt gebruikt om een element uit de dataset in op te slaan. Een element kan bijvoorbeeld een afbeelding zijn: in dat geval slaan we hiervan de grijswaarden van de pixels op. Het doel van het Neurale Netwerk is classificatie: ieder element uit de dataset heeft een klasse, welke we correct willen leren voorspellen. De datastructuur wordt gebruikt om de activatiewaarden van de inputlaag, via de gewogen kanten in het netwerk, door te geven naar de laatste laag. Dit noemt men feed-forward, en gebeurt door het matrix-vector product te nemen van de voorgaande vector en de bijbehorende gewichtenmatrix.

Vaak wordt in aanvulling op het matrix-vector product nog een activatiefunctie toegepast op iedere vector. Dit wordt gebruikt om de activatiewaarden van de neuronen te schalen of te normaliseren. Een gebruikelijke keuze hierin is de sigmoïde functie, $f(x) = \frac{1}{1+e^{-x}}$.

Definitie 4.1.2. *Gegeven een Neuraal Netwerk N en activatiefunctie $f(x)$, dan wordt de waarde van vector $\mathbf{x}_{m+1}$ berekend door herhaaldelijk toepassen van:*

$$\mathbf{x}_k(i) = f\left(\sum_{j=1}^{|\mathbf{x}_{k-1}|} \mathbf{W}_k(i,j) * \mathbf{x}_{k-1}(j)\right)$$

voor $1 \leq i \leq |\boldsymbol{x}_k|$ en voor $1 \leq k \leq m+1$

We kunnen dit compacter weergeven door te noteren: $\mathbf{x}_k = f(\mathbf{W}_k\mathbf{x}_{k-1})$.

In definitie 4.1.1 is te zien dat de gewichtenmatrices al de juiste formaten hebben voor deze berekeningen. Na afloop van deze stap heeft iedere output neuron $x_{m+1}(i)$ een waarde, welke als volgt worden geïnterpreteerd: iedere neuron representeert een mogelijke *klasse*, en de bijbehorende activatiewaarde representeert de score van de ingevoerde vector in die gegeven klasse. De neuron met de hoogste classificatiescore wordt door het programma opgeslagen als *de geschatte klasse voor het ingevoerde element.* Uiteindelijk moet deze classificatie zo accuraat mogelijk worden uitgevoerd, d.w.z.: dat de geschatte klasse voor zo veel mogelijk elementen overeenkomt met de daadwerkelijke klasse.

4.2 Leren met back-propagation

In het voorgaande gedeelte hebben we gezien op welke manier een neuraal netwerk een enkel element uit een dataset kan classificeren. De classificatiescore is volledig afhankelijk van de gewichten in het netwerk. Dit hoofdstuk gaat over een manier waarop deze gewichten iteratief kunnen worden aangepast zodat de classificatie steeds accurater wordt: dit is een proces dat gezien kan worden als leren. De techniek die hiervoor gebruikt wordt is backpropagation. Deze techniek wordt al sinds 1960 onderzocht en kent veel variaties: de variant die in deze scriptie wordt gebruikt, zoals onder andere beschreven in [6], is de meest gebruikelijke. Het idee van back-propagation is dat ieder gewicht wordt gewijzigd naar de mate waarin deze heeft bijgedragen aan de totale foutmarge. De fout per output neuron $\mathbf{x}_{m+1}(i)$ slaan we op in de error vector $\mathbf{E}$. De foutfunctie E is vervolgens gegeven als de som van alle elementen in $\mathbf{E}$:

$$\mathbf{E}(i) = \frac{1}{2}(\mathbf{x}_{m+1}(i) - t_i)^2 \quad \text{voor} \quad 1 \leq i \leq |\mathbf{x}_{m+1}|$$

$$E = \sum_{i=1}^{|\mathbf{x}_{m+1}|} \mathbf{E}(i)$$

Hierin is t_i de doelwaarde van de neuron $\mathbf{x}_{m+1}(i)$. Deze doelwaarde is 1 indien het ingevoerde element tot de klasse behoort waar de gegeven neuron voor staat, en 0 anders. We willen E minimaliseren tijdens het leerproces. Dit doen we door voor ieder gewicht w de partiële afgeleide te nemen van de foutfunctie met als variabele dat gewicht w. Deze partiële afgeleide, welke we noteren als $\frac{\partial E}{\partial w}$, wordt vervolgens gebruikt om het gewicht aan te passen als volgt:

$$w = w - \alpha \frac{\partial E}{\partial w} \quad \text{voor alle} \quad w \in \mathbf{W}_k, \quad 1 \leq k \leq m+1 \tag{1}$$

De waarde α staat hier voor een scalar die gebruikt wordt om de snelheid van het leerproces te beïnvloeden. De vergelijking die we nodig hebben voor de berekening van de partiële afgeleide kunnen we vinden door herhaaldelijk de kettingregel toe te passen. We beginnen bij de laatste gewichtenmatrix $\mathbf{W}_{m+1}$. We leiden af voor alle i, j, met $1 \leq i \leq |\mathbf{x}_{m+1}|$ en $1 \leq j \leq |\mathbf{x}_m|$:

$$\frac{\partial E}{\partial \mathbf{W}_{m+1}(i,j)} = \frac{\partial E}{\partial \mathbf{x}_{m+1}(i)} \frac{\partial \mathbf{x}_{m+1}(i)}{\partial \mathbf{W}_{m+1}\mathbf{x}_m(i)} \frac{\partial \mathbf{W}_{m+1}\mathbf{x}_m(i)}{\partial \mathbf{W}_{m+1}(i,j)} \tag{2}$$

De eerste stap die we hierboven zetten is differentiëren naar het i^e element in de output vector $\mathbf{x}_{m+1}$. Het gewicht $\mathbf{W}_{m+1}(i,j)$ heeft immers alleen betrekking op $\mathbf{x}_{m+1}(i)$, volgens definitie 4.1.2. Met de definitie van E vinden we vervolgens dat $\frac{\partial E}{\partial \mathbf{x}_{m+1}(i)} = \mathbf{x}_{m+1}(i) - t_i$, aangezien in E alle overige elementen 0 worden na differentiatie naar $\mathbf{x}_{m+1}$.

Vervolgens gebruiken we de definitie van $\mathbf{x}_{m+1}(i)$ zoals in definitie 4.1.2. De sommatie geven we weer als $\mathbf{W}_{m+1}\mathbf{x}_m(i)$. Hiermee bedoelen we: het i^e element uit de vector die resulteert wanneer we het matrix-vector product nemen van $\mathbf{W}_{m+1}$ en $\mathbf{x}_m$. Daardoor we kunnen schrijven: $\mathbf{x}_{m+1}(i) = f(\mathbf{W}_{m+1}\mathbf{x}_m(i))$. De volgende stap in de differentiatie is dus de afgeleide nemen van de activatiefunctie, waardoor we vinden $\frac{\partial \mathbf{x}_{m+1}(i)}{\partial \mathbf{W}_{m+1}\mathbf{x}_m(i)} = f'(\mathbf{W}_{m+1}\mathbf{x}_m(i))$.

Tot slot vinden we $\frac{\partial \mathbf{W}_{m+1}\mathbf{x}_m(i)}{\partial \mathbf{W}_{m+1}(i,j)} = \mathbf{x}_m(j)$, hetgeen volledig af te leiden is uit de definitie van het matrixproduct.

We vinden dus

$$\frac{\partial E}{\partial \mathbf{W}_{m+1}(i,j)} = (\mathbf{x}_{m+1}(i) - t_i) * f'(\mathbf{W}_{m+1}\mathbf{x}_m(i)) * \mathbf{x}_m(j) \tag{3}$$

We willen nu eenzelfde soort afleiding uitvoeren voor de overige gewichtenmatrices. We zullen ontdekken dat dit een iteratief proces is vanaf $\mathbf{W}_m$ tot en met $\mathbf{W}_1$. Hierin kunnen we gebruik maken van tussenresultaten, namelijk de gradiënt δ_k van de foutfunctie E met betrekking tot laag $\mathbf{W}_k\mathbf{x}_{k-1}$. Dit is een vector van lengte $|\mathbf{x}_k|$, welke gegeven is door:

$$\delta_k = \frac{\partial E}{\partial \mathbf{W}_k\mathbf{x}_{k-1}} \quad \text{voor} \quad 1 \leq k \leq m+1$$

$$\text{waarin} \quad \delta_k(i) = \frac{\partial E}{\partial \mathbf{W}_k\mathbf{x}_{k-1}(i)} \quad \text{voor} \quad 1 \leq i \leq |\mathbf{x}_k|$$

De definitie van δ_{m+1} is reeds af te leiden vanuit vergelijking (2), namelijk als volgt:

$$\delta_{m+1}(i) = \frac{\partial E}{\partial \mathbf{x}_{m+1}(i)} \frac{\partial \mathbf{x}_{m+1}(i)}{\partial \mathbf{W}_{m+1}\mathbf{x}_m(i)} \quad \text{, voor} \quad 1 \leq i \leq |\mathbf{x}_{m+1}| \tag{4}$$

$$= (\mathbf{x}_{m+1}(i) - t_i) * f'(\mathbf{W}_{m+1}\mathbf{x}_m(i)) \tag{5}$$

Bovenstaande kunnen we nu gebruiken om de afgeleide van E te berekenen voor een gewicht $\mathbf{W}_k(i,j)$ in een van de overige lagen k indien δ_{k+1} bekend is. Dit leiden we als volgt af, voor een willekeurige k, $1 \leq k \leq m$, en voor alle i, j, $1 \leq i \leq |x_{k+1}|$, $1 \leq j \leq |x_k|$:

$$\frac{\partial E}{\partial \mathbf{W}_k(i,j)} = \frac{\partial E}{\partial \mathbf{x}_k(i)} \frac{\partial \mathbf{x}_k(i)}{\partial \mathbf{W}_k\mathbf{x}_{k-1}(i)} \frac{\partial W_k\mathbf{x}_{k-1}(i)}{\partial \mathbf{W}_k(i,j)} \tag{6}$$

Het beschouwde gewicht $\mathbf{W}_k(i,j)$ heeft invloed op $\mathbf{x}_k(i)$, en $\mathbf{x}_k(i)$ heeft invloed op alle elementen in $\mathbf{x}_{k+1}$. Daarom is het nodig om de kettingregel toe te passen voor ieder element l in $\mathbf{x}_{k+1}$ om de afgeleide te vinden van E naar $\mathbf{x}_k(i)$. Dit gaat als volgt:

$$\frac{\partial E}{\partial \mathbf{W}_k(i,j)} = \left[\sum_{l=1}^{|\mathbf{x}_{k+1}|} \frac{\partial E}{\partial \mathbf{x}_{k+1}(l)} \frac{\partial \mathbf{x}_{k+1}(l)}{\partial \mathbf{W}_{k+1}\mathbf{x}_k(l)} \frac{\partial \mathbf{W}_{k+1}\mathbf{x}_k(l)}{\partial \mathbf{x}_k(i)} \right] \frac{\partial \mathbf{x}_k(i)}{\partial \mathbf{W}_k\mathbf{x}_{k-1}(i)} \frac{\partial W_k\mathbf{x}_{k-1}(i)}{\partial \mathbf{W}_k(i,j)}$$

Om te beginnen merken we op: $\frac{\partial E}{\partial \mathbf{x}_{k+1}(l)} \frac{\partial \mathbf{x}_{k+1}(l)}{\partial \mathbf{W}_{k+1}\mathbf{x}_k(l)} = \frac{\partial E}{\partial \mathbf{W}_{k+1}\mathbf{x}_k(l)} = \delta_{k+1}(l)$. Hier zien we dus dat $\frac{\partial E}{\partial \mathbf{W}_k(i,j)}$ berekend kan worden door gebruik te maken van de gradiënt van E met betrekking tot laag $\mathbf{W}_{k+1}\mathbf{x}_k$.

Vervolgens leiden we af: $\frac{\partial \mathbf{W}_{k+1}\mathbf{x}_k(l)}{\partial \mathbf{x}_k(i)} = \mathbf{W}_{k+1}(l,i)$, volledig volgens de definitie van het matrixproduct.

Daarna schrijven we weer $\mathbf{x}_k(i) = f(\mathbf{W}_k\mathbf{x}_{k-1}(i))$, zoals we ook deden met $\mathbf{x}_{m+1}$ voor vergelijking (2). Hieruit volgt dat de volgende stap het differentiëren van de activatiefunctie is: $\frac{\partial \mathbf{x}_k(i)}{\partial \mathbf{W}_k\mathbf{x}_{k-1}(i)} = f'(\mathbf{W}_k\mathbf{x}_{k-1}(i))$.

Tot slot vinden we weer dat $\frac{\partial W_k\mathbf{x}_{k-1}(i)}{\partial \mathbf{W}_k(i,j)} = x_{k-1}(j)$ met de definitie van het matrixproduct.

Het resultaat is als volgt:

$$\frac{\partial E}{\partial \mathbf{W}_k(i,j)} = \left[\sum_{l=1}^{|\mathbf{x}_{k+1}|} \delta_{k+1}(l)\mathbf{W}_{k+1}(l,i) \right] * f'(\mathbf{W}_k\mathbf{x}_{k-1}(i)) * x_{k-1}(j) \tag{7}$$

Zoals reeds opgemerkt hebben we voor bovenstaande berekening δ_{k+1} nodig. Aangezien tot nu toe slechts δ_{m+1} bekend is, zou dat betekenen dat we niet verder komen dan de afleiding van $\frac{\partial E}{\partial \mathbf{W}_m}$. Echter laten we hieronder zien hoe het mogelijk is om voor willekeurige k de bijbehorende δ_k af te leiden aan de hand van de vergelijking van $\frac{\partial E}{\partial \mathbf{W}_k}$:

$$\delta_k(i) = \frac{\partial E}{\partial \mathbf{W}_k\mathbf{x}_{k-1}(i)} = \frac{\partial E}{\partial \mathbf{x}_k(i)} \frac{\partial \mathbf{x}_k(i)}{\partial \mathbf{W}_k\mathbf{x}_{k-1}(i)}$$

$$= \left[\sum_{l=1}^{|\mathbf{x}_{k+1}|} \delta_{k+1}(l)\mathbf{W}_{k+1}(l,i) \right] * f'(\mathbf{W}_k\mathbf{x}_{k-1}(i)) \quad \text{voor} \quad 1 \leq i \leq |\mathbf{x}_k|$$

Bovenstaande vergelijking kan in matrixnotatie worden weergegeven, als volgt:

$$\begin{aligned}\delta_k &= \left[\delta_{k+1}^\top \mathbf{W}_{k+1}\right]^\top \circ f'\left(\mathbf{W}_k \mathbf{x}_{k-1}\right) \\ &= \left[\mathbf{W}_{k+1}^\top \delta_{k+1}\right] \circ f'\left(\mathbf{W}_k \mathbf{x}_{k-1}\right)\end{aligned}$$

Waarin $\circ$ staat voor het elementsgewijs vermenigvuldigen.

Alle onderdelen die nodig zijn voor de berekening van (1) zijn nu gegeven. Resumerend geven we de vergelijkingen voor de back-propagation als volgt weer in matrixnotatie:

$$\delta_{m+1} = (\mathbf{x}_{m+1} - t) \circ f'(\mathbf{W}_{m+1}\mathbf{x}_m) \tag{8}$$

$$\delta_k = \left[\mathbf{W}_{k+1}^\top \delta_{k+1}\right] \circ f'\left(\mathbf{W}_k \mathbf{x}_{k-1}\right) \quad \text{voor} \quad 1 \leq k \leq m \tag{9}$$

$$\frac{\partial E}{\partial \mathbf{W}_k} = \delta_k \mathbf{x}_{k-1}^\top \quad \text{voor} \quad 1 \leq k \leq m+1 \tag{10}$$

Zoals gezegd wordt de berekening iteratief uitgevoerd, beginnend bij $\mathbf{W}_{m+1}$ tot en met $\mathbf{W}_1$. Eerst worden de waarden van de δ's berekend en vervolgens worden daarmee de gewichtenmatrices aangepast. Door de manier waarop deze aanpassing tot stand komt weten we zeker dat hierdoor de foutfunctie E afneemt. Daaruit volgt dat de geschatte klasse van ieder element dichter in de buurt komt van de daadwerkelijke klasse van het element, hetgeen het doel is van de back-propagation. Bovenstaande operatie wordt doorgaans meerdere malen toegepast op ieder element in een omvangrijke dataset: de hoop is dat het model dan convergeert naar een model dat voor ieder willekeurig element nauwkeurig de klasse kan voorspellen.

4.3 Toepassing: cijferherkenning

In het programma dat voor deze scriptie is geschreven wordt een Neuraal Netwerk toegepast voor het herkennen van handgeschreven getallen. Dit is een veel gebruikte eerste oefening op het gebied van classificatie en er is een grote dataset beschikbaar om een model mee te trainen[7]. In dit gedeelte wordt kort uitgelegd hoe dit probleem kan worden aangepakt met een Neuraal Netwerk.

We beginnen hierin bij de input: dit zijn de elementen uit de dataset. Een element is een afbeeldingen van 28×28 pixels, welke een getal tussen 0 en 9 voorstelt. De afbeeldingen zijn zwart-wit, waardoor de opslag gebeurt door voor iedere pixel de grijswaarde op te slaan. Als input vector volstaat dus een vector van 784×1 met op iedere positie de grijswaarde van een pixel.

De neuronen in de output laag stellen de verschillende klassen voor: in dit geval zijn dat de getallen 0 t/m 9. Het doel van het Neurale Netwerk is het correct voorspellen van de klasse van de elementen uit de dataset: op die manier kunnen we controleren of het model een handgeschreven getal kan 'herkennen'.

De hoeveelheid verborgen lagen, de grootte van de verschillende verborgen lagen, de activatiefunctie en de waarde van de leersnelheid α zijn variabelen die door de gebruiker gekozen kunnen worden. Een groot aantal verborgen lagen zorgt bij dit probleem onnodig voor meer complexiteit, waardoor er in de praktijk vaak wordt gekozen voor 1 of 2 lagen met 30 tot 600 neuronen. Als activatiefunctie wordt vaak de sigmoïde functie gebruikt, welke als voordeel heeft dat de afgeleide een gemakkelijke vergelijking heeft: $f'(x) = f(x)(1 - f(x))$.

4.4 Uitbreiding: ijlheid

Traditioneel zijn de verschillende lagen in een neuraal netwerk all-to-all verbonden. Dat betekent dat er voor iedere neuron in een verborgen laag een gewogen connectie met iedere neuron in de voorgaande en opvolgende laag is. De gewichtenmatrices zijn hierdoor *dicht*. Deze eigenschap is echter niet van cruciaal belang voor de werking van het Neurale Netwerk zoals hierboven beschreven. In dit gedeelte komt een aantal argumenten aan bod voor het opzetten van een Neuraal Netwerk met *ijle* gewichtenmatrices.

Allereerst grijpen we terug op de biologische achtergrond van het Neurale Netwerk: het brein. Het netwerk van neuronen waardoor in het menselijk en dierlijk brein impulsen worden doorgegeven kan worden geanalyseerd als een complexe graaf [8]. Dit leert ons dat er sprake is van clustering en 'small-world topology'. Het brein lijkt dus totaal geen all-to-all eigenschap te hebben, hetgeen de eerste aanleiding is om onderzoek te doen naar een Neuraal Netwerk dat die eigenschap ook niet heeft.

Wanneer we het Neurale Netwerk dan beschouwen als een graaf waarin niet iedere laag compleet verbonden is, merken we op dat de gewichtenmatrices ijler worden. Dit zorgt ervoor dat zowel de operaties voor de feed-forward als de back-propagation met minder variabelen moeten rekenen. Dit zou in theorie veel rekentijd kunnen schelen, hetgeen bij Neurale Netwerken een bijzonder belangrijke factor is.

Vervolgens merken we op dat, met behulp van bijvoorbeeld de GraphBLAS, *ijle* matrices efficiënter kunnen worden opgeslagen. Dit wordt gedaan door gebruik te maken van een 'structural zero', of door de waarden op te slaan als tuples. Hierdoor neemt de hoeveelheid computergeheugen benodigd voor de opslag en de berekeningen af. Daarnaast lenen operaties op ijle matrices zich beter voor parallellisatie, wat wederom voor een afname van rekentijd kan zorgen. Ook deze optimalisatie wordt ondersteund door de GraphBLAS.

Een voor de hand liggende operatie voor het opleggen van ijlheid aan de gewichtenmatrices is het verwijderen van bepaalde gewichten na afloop van de training. We nemen immers aan dat de gewichten dan zijn geconvergeerd, zodanig dat de classificatie zorgt voor een minimale error. Vermenigvuldigen van een activatiewaarde met een gewicht *in de buurt van* 0 heeft haast geen invloed op de waarde in de opvolgende laag. Als we die kleine gewichten verwijderen uit het model laten we de classificatie over aan de gewichten die al een grote invloed hadden op de classificatie. De verwachting is dat het model op die manier weinig nauwkeurigheid verliest.

5 Implementatie

Dit hoofdstuk gaat over de code die voor deze scriptie is geschreven. Hierbij zullen de gemaakte keuzes worden toegelicht en volgt er een overzicht van het programma in pseudocode. Tot slot volgt een overzicht van de resultaten van de verschillende tests die met het programma zijn uitgevoerd.

5.1 Implementatie in GraphBlas

Het geschreven programma implementeert een Neuraal Netwerk dat gebruik maakt van de functies van de GraphBLAS. Het programma is geschreven in de

taal C++. We maken daarbij gebruik van een algemeen beschikbare *header-only library* [9], waarin de verschillende functies en objecten van de GraphBLAS zijn gedefinieerd.

Het programma bestaat grofweg uit 3 operaties: Constructie, classificatie en training. De constructie initieert eerst de vectoren waarin de activatiewaarden worden opgeslagen. Aangezien het netwerk gebruikt wordt voor de cijferclassificatie bestaat het model uit een inputvector met dimensie 784×1, een verborgen vector met dimensie 30×1 en een outputvector met dimensie 10×1. Doordat we gebruik maken van slechts 1 verborgen laag convergeert het model snel en is de nauwkeurigheid voldoende. Vervolgens worden 2 gewichtenmatrices van het juiste formaat geïnitieerd en gevuld met willekeurige gewichten. De eerste gewichtenmatrix heeft dimensies 30×784, de tweede 10×30. Om het vullen van de gewichtenmatrices snel uit te kunnen voeren is gekozen om de willekeurige gewichten in stappen van 0,2 tussen -1 en 1 te kiezen. Voor de opslag van de matrices en vectoren wordt gebruikt gemaakt van het **CscMatrix** object aangezien zo kan worden geprofiteerd van eventuele ijlheid.

De classificatie wordt uitgevoerd door het nemen van het matrix-vector product en het toepassen van de sigmoïde functie, zoals beschreven in hoofdstuk 4.1. Hiervoor worden de GraphBLAS-functies **MxV** en **Apply** gebruikt.

Het algoritme voor de training implementeert de back-propagation. Hiervoor classificeren we eerst een element en passen we aan de hand van de foutfunctie de gewichten aan. Dit gaat zoals beschreven in hoofdstuk 4.2: de eerste stap is het opstellen van de δ behorende bij de output door het vermenigvuldigen van de fout per neuron met de afgeleide van de activatiefunctie. Daarna wordt de voorgaande δ berekend door het nemen van het matrixproduct met de laatste gewichtenmatrix en het elementsgewijs vermenigvuldigen met de afgeleide van de activatiefunctie van de voorgaande laag. Met die δ's kunnen vervolgens de gewichtenmatrices worden aangepast. De operaties zijn gebaseerd op vergelijkingen (8), (9) en (10) uit hoofdstuk 4.2. De implementatie maakt gebruik van de GraphBLAS-functies **MxM**, **Apply**, **Transpose** en **EwiseMult**.

5.2 Pseudocode

Hieronder volgt een beschrijving van de functies voor classificatie en training in pseudocode. In het algoritme dat de training implementeert is het van belang dat de waarden van de verborgen vector H uitgelezen kunnen worden: daarom worden deze tijdens het algoritme dat de classificatie uitvoert opgeslagen in het geheugen. De volledige code van het neurale netwerk is in Appendix A te vinden.

Algoritme 1: Classificatie

```
1  using graphblas
2  functie Feed-forward (I, W1, W2)
       Input  : inputvector I, gewichtenmatrices W1,W2
       Output: outputvector O
3          vector H ← graphblas.MxV(W1, I)
4          graphblas.Apply(H, sigmoïde)
5          vector O ← graphblas.MxV(W2, H)
6          graphblas.Apply(O, sigmoïde)
7          return O
8  functie sigmoïde (a)
       Input  : scalar a
       Output: scalar f(a) = 1/(1+e^-a)
```

Algoritme 2: Training

```
1  using graphblas
2  functie Back-propagation (I, t, W1, W2)
       Input  : inputvector I, klasse t, gewichtenmatrices W1,W2
       Output: Nieuwe gewichtenmatrices W1', W2'
3          O ← Feed-forward(I, W1, W2)
4          δ2 ← vector(|O|)
5          for elke index i ∈ δ2:
6              if i = t:
7                  δ2[i] = (1 − O[i])∗sigmoïdeAfgeleide(O[i])
8              else:
9                  δ2[i] = (O[i])∗sigmoïdeAfgeleide(O[i])
10         δ1 ← graphblas.MxM(Transpose(W2), δ2)
11         graphblas.Apply(H, sigmoïdeAfgeleide)
12         graphblas.EwiseMult(δ1, H)
13         W2 ← W2 + graphblas.MxM(δ2, Transpose(H))
14         W1 ← W1 + graphblas.MxM(δ1, Transpose(I))
15         Return W1, W2
16 functie sigmoïdeAfgeleide (a)
       Input  : scalar a
       Output: scalar f'(a) = a(1 − a)
```

5.3 Resultaten

In dit hoofdstuk wordt een overzicht gegeven van de resultaten van verschillende tests die zijn uitgevoerd met het geïmplementeerde Neurale Netwerk. Deze tests zijn erop gericht de prestaties van het netwerk te analyseren bij variërende ijlheid in de matrices en vectoren.

Allereerst is de invloed van ijlheid in de inputvector op de rekentijd onderzocht. Dit is gedaan door een classificatiestap en een trainingsstap uit te voeren met inputvectoren van variërende dichtheid. Hieronder zijn de resultaten daarvan weergegeven in een tabel: in de linkerkolom is de hoeveelheid gewichten aangegeven en in de bijbehorende rij is per operatie de tijd genoteerd.

$\|x_0\|$	(1)	(2)	(3)	(4)	(5)	(6)
784	75,9 ms	73,55 ms	2,3 ms	9864,3 ms	9691,1 ms	19,2 ms
560	63,65 ms	61,3 ms	2,3 ms	5691,9 ms	5500,7 ms	19,2 ms
392	50,44 ms	48,09 ms	2,3 ms	3033,3 ms	2868,1 ms	19,2 ms
168	34,46 ms	32,11 ms	2,3 ms	940,4 ms	848,2 ms	19,2 ms
56	26,9 ms	24,55 ms	2,3 ms	458,3 ms	393,1 ms	19,2 ms

Figuur 5: Rekentijden van verschillende operaties in het neurale netwerk met variabel aantal elementen in de inputvector

De tests zijn uitgevoerd met gewichtenmatrices die volledig gevuld zijn, zodat het effect van de ijlheid in slechts de inputvector te analyseren is. De operaties waarvan de tijd gemeten is, zijn: de volledige classificatiestap (1), het matrixproduct van de inputvector met de eerste gewichtenmatrix (2), het matrixproduct van de eerste verborgen laag met de tweede gewichtenmatrix (3), de volledige trainingsstap (4), het matrixproduct voor het aanpassen van de eerste (5) en tweede (6) gewichtenmatrix gegeven de juiste delta.

Vervolgens is de invloed onderzocht van ijlheid in de eerste en tweede gewichtenmatrix. Hiervoor is enkel de classificatiestap uitgevoerd met een volledig gevulde inputvector en een variërende hoeveelheid gewichten in de matrices W1 en W2. De trainingsstap is niet uitgevoerd aangezien dat invloed zou hebben op de hoeveelheid gewichten en daarmee de ijlheid zou verdwijnen. De resultaten zijn hieronder te vinden:

$\%w$	(1)	(2)	(3)
100%	75,9 ms	73,55 ms	2,3 ms
75%	64,46 ms	62,11 ms	2,3 ms
50%	52,13 ms	49,78 ms	2,3 ms
25%	37,73 ms	35,38 ms	2,3 ms
10%	29,15 ms	26,8 ms	2,3 ms
0%	19,75 ms	17,4 ms	2,3 ms

Figuur 6: Rekentijden van verschillende operaties in het neurale netwerk met variabel aantal gewichten als percentage van de totale grootte van de gewichtenmatrices

Tot slot is de invloed op de nauwkeurigheid onderzocht wanneer na een serie trainingsstappen, waarin het netwerk is geconvergeerd, een reductie wordt uitgevoerd op de gewichtenmatrices. Dit is gedaan door per gewichtenmatrix de gewichten kleiner dan een gegeven drempelwaarde te verwijderen. Daarnaast worden ook de gewichten verwijderd die niet zijn aangepast tijdens de gehele trainingscyclus, aangezien deze dan ook geen invloed hebben op de classificatie. De gebruikte drempelwaarde staat in de linkerkolom: in de bijbehorende rij is de hoeveelheid resterende gewichten en de nauwkeurigheid weergegeven. De test is tweemaal uitgevoerd voor een variabel aantal trainingselementen en drempelwaardes. De nauwkeurigheid is bepaald door 1000 elementen te classificeren die niet in de trainingsset voorkwamen.

Reductie	$\%\mathbf{W}_1$	$\%\mathbf{W}_2$	Nauwkeurigheid
	100%	100%	84.1%
$\|w\| < 0.3$	58%	100%	84.7%
$\|w\| < 0.3$	58%	78%	84.4%
$\|w\| < 0.5$	41%	78%	83.8%
$\|w\| < 0.7$	25%	78%	82.6%
$\|w\| < 0.5$	25%	62%	83%

Figuur 7: Nauwkeurigheid van classificatie na tweemaal training op 3000 elementen en reductie op $\mathbf{W}_1$, $\mathbf{W}_2$, $\mathbf{W}_1$, $\mathbf{W}_1$ en $\mathbf{W}_2$

Reductie	$\%\mathbf{W}_1$	$\%\mathbf{W}_2$	Nauwkeurigheid
	100%	100%	88%
$\|w\| < 0.35$	59%	100%	87.8%
$\|w\| < 0.85$	13%	100%	86.5%
$\|w\| < 0.35$	13%	83%	86.5%
$\|w\| < 0.85$	13%	43%	87%

Figuur 8: Nauwkeurigheid van classificatie na tweemaal training op 6000 elementen en reductie op $\mathbf{W}_1$, $\mathbf{W}_1$, $\mathbf{W}_2$ en $\mathbf{W}_2$

6 Conclusie en Discussie

In dit laatste hoofdstuk wordt teruggegrepen op het doel van de scriptie en worden de onderzoeksresultaten geïnterpreteerd. Het doel van het onderzoek was het programmeren en testen van een Neuraal Netwerk dat gebruik maakt van de functies en objecten van de GraphBLAS. Specifiek waren we benieuwd naar de prestaties wanneer door middel van reductie in de gewichtenmatrices en vectoren voor ijlheid wordt gezorgd. De prestaties van een Neuraal Netwerk worden op 2 gebieden beoordeeld: rekentijd en nauwkeurigheid.

Onze eerste observatie met betrekking tot de rekentijd is dat deze bijna volledig in beslag wordt genomen door het uitvoeren van de berekeningen die betrekking hebben op de eerste gewichtenmatrix. Met name het updaten van die gewichtenmatrix (operatie (5) uit hoofdstuk 5.3) is intensief: dat komt waarschijnlijk doordat deze berekening resulteert in 784×30 outputwaarden. We merken op dat van beide operaties de rekentijd sterk vermindert indien er meer ijlheid ontstaat: de bijna lineaire afname wijst erop dat er optimaal geprofiteerd wordt van het ontbreken van elementen.

Ten tweede observeren we de nauwkeurigheid van de classificatie na de uitgevoerde reductie. Deze resultaten zijn bijzonder positief: de nauwkeurigheid blijft gemiddeld hoog, zelfs nadat het overgrote deel van de gewichten is verwijderd. Dit betekent dat de classificatie vooral afhankelijk is van de gewichten met een grote absolute waarde in het netwerk, en dat de gewichten met kleine waarde minder relevant zijn dan doorgaans wordt aangenomen.

Bovenstaande observaties bieden nieuwe mogelijkheden betreffende de architectuur van Neurale Netwerken. Duidelijk is dat de all-to-all architectuur geen

vereiste is voor het gemiddeld goed presteren van Neurale Netwerken. Meer onderzoek is nodig om uit te wijzen of hetzelfde geldt voor implementaties die voor andere doeleinden worden gebruikt dan het herkennen van handgeschreven getallen. Ook kan er nog meer werk worden verricht op het gebied van de optimalisatie van de rekentijd: het matrixproduct voor ijle matrices is hierin een vereiste. De GraphBLAS laat duidelijk zien dat hiermee veel winst te behalen is. Tot slot is de parallellisatie van de matrixoperaties niet aan bod gekomen aangezien dit niet binnen de focus van de scriptie lag. Ook op dat gebied is nog onderzoek mogelijk, zeker ook in combinatie met de GraphBLAS: dit is immers een standaard die zich daar in theorie goed voor leent.

Resumerend kunnen we stellen dat de implementatie van het Neurale Netwerk met behulp van de GraphBLAS succesvol is en dat de resultaten van de tests hebben geleid tot nieuwe inzichten. Dit spoort aan tot verder onderzoek naar optimalisatie van Neurale Netwerken door ijlheid, en voor de implementatie hiervan lijkt de GraphBLAS een veelbelovende kandidaat.

Referenties

[1] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, 2011.

[2] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. Mathematical foundations of the GraphBLAS. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pages 1–9. IEEE, 2016.

[3] Aydin Buluç, Timothy Mattson, Scott McMillan, José Moreira, and Carl Yang. The GraphBLAS C API Specification. Technical report, Combinatorial Scientific Computing Lab at the University of California Santa Barbara., 2017.

[4] An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Softw.*, 28(2):135–151, June 2002.

[5] Aydin Buluç and John Gilbert. The Combinatorial BLAS: Design, Implementation and Applications. http://www.cs.ucsb.edu/research/tech_reports/reports/2010-18.pdf.

[6] https://www.cs.swarthmore.edu/ meeden/cs81/s10/BackPropDeriv.pdf.

[7] http://yann.lecun.com/exdb/mnist/.

[8] Ed Bullmore and Olaf Sporns. Complex brain networks: graph theoretical analysis of structural and functional systems. *Nat Rev Neurosci*, 10(3):186–198, Mar 2009.

[9] https://github.com/cmu-sei/gbtl.

A C++ Code Neuraal Netwerk

```
#include <iostream>
#include <ctime>
#include <graphblas/graphblas.hpp>
#include <string>
#include <sstream>
#include <vector>
#include <fstream>
#include <iomanip>
#include <chrono>
typedef std::chrono::high_resolution_clock Clock;
typedef double ScalarType;

//Utility function defining the sigmoid transformation of
    a scalar and it's derivative
ScalarType sigmoid(ScalarType a) {
  return 1 / (1 + exp(-a));
}

ScalarType deriv(ScalarType a) {
  return a*(1 - a);
}

//Function for applying learning rate to scalar
ScalarType learn_rate_reduce(ScalarType a) {
  return a*0.35;
}
ScalarType learn_rate_enlarge(ScalarType a) {
  return a * 1.5;
}

//Functions for reducing elements
ScalarType reduce_a(ScalarType a) {
  if (a < 0.93 && a > -0.93) return 0;
  else if (a==0.2|| a==0.4 || a == 0.6 || a == 0.8 || a
     == 1 || a == -0.2 || a == -0.4 || a == -0.6 || a ==
      -0.8 || a == -1) return 0;
  return a;
}
ScalarType reduce_b(ScalarType a) {
  if (a < 1 && a > -1) return 0;
  return a;
}
ScalarType reduce_c(ScalarType a) {
  if (a < 1.1 && a > -1.1) return 0;
  return a;
}
```

```
class NeuralNet {
  public:
    //Elements defining the neural net
    graphblas::CscMatrix<ScalarType> hidden_one,
        hidden_two, output, weight_two,
                    delta_one, delta_out, weight_two_t,
                    input, weight_one, weight_one_t;
    graphblas::CsrMatrix<ScalarType> input_t,
        hidden_one_t, hidden_two_t;
    int learningRate;
    double totalError;
    int numHidden;

    //Default constructor
    NeuralNet::NeuralNet(void) : input(1,1), hidden_one
        (1,1), hidden_two(1,1), output(1,1), weight_one
        (1,1), weight_two(1,1), delta_one(1,1), delta_out
        (1,1), weight_one_t(1,1), weight_two_t(1, 1),
        input_t(1, 1), hidden_one_t(1, 1), hidden_two_t(1,
         1){};

    //Function for creating network-instance with
        dimensions
    void Fill(int Num_inputs, int Num_hidden_one, int
        Num_outputs) {
      input = graphblas::CscMatrix<ScalarType>(Num_inputs
          , 1);
      input_t = graphblas::CsrMatrix<ScalarType>(1,
          Num_inputs);
      hidden_one = graphblas::CscMatrix<ScalarType>(
          Num_hidden_one, 1);
      delta_one = graphblas::CscMatrix<ScalarType>(
          Num_hidden_one, 1);
      hidden_one_t = graphblas::CsrMatrix<ScalarType>(1,
          Num_hidden_one);
      output = graphblas::CscMatrix<ScalarType>(
          Num_outputs, 1);
      delta_out = graphblas::CscMatrix<ScalarType>(
          Num_outputs, 1);

      //Instantiate and construct weight matrices
      weight_one = buildRandCscMatrix(Num_hidden_one,
          Num_inputs);
      weight_one_t = graphblas::CscMatrix<ScalarType>(
          Num_inputs, Num_hidden_one);
      weight_two = buildRandCscMatrix(Num_outputs,
          Num_hidden_one);
      weight_two_t = graphblas::CscMatrix<ScalarType>(
          Num_hidden_one, Num_outputs);
    }
```

```
//Function for filling a matrix of given dimensions
    with weights between -1 and 1
graphblas::CscMatrix<ScalarType> buildRandCscMatrix(
    int rows, int cols) {
  std::vector<std::vector<ScalarType>>  z(rows);
  for (int i = 0; i < rows; i++) {
    z[i] = std::vector<ScalarType>(cols);
    for (int x = 0; x < cols; x++) {
       z[i][x] = ((rand() % 10 - 5) / 5.0) + 0.05;}
  }
  graphblas::CscMatrix<ScalarType> aMatrix(z, 0.0);
  return aMatrix;
}

//Function for the classification of an input.
    Returns the class index.
int Classify(std::vector<ScalarType> element) {
  //Fill input vector as CSC matrix
  std::vector<std::vector<ScalarType>> elem(784);
  for (int i = 0; i < 784; i++) {
    elem[i] = { element[i] };
  }
  input = graphblas::CscMatrix<ScalarType>(elem, 0.0)
      ;

  //Needed for backend call
  graphblas::ArithmeticSemiring<ScalarType> s;
  graphblas::math::Assign<ScalarType> assign;
  graphblas::math::Accum<ScalarType> add;

  //Multiply and apply sigmoid
  graphblas::backend::mxv(weight_one, input,
      hidden_one, s, assign);
  graphblas::backend::apply(hidden_one, hidden_one,
      sigmoid, assign);
  graphblas::backend::mxv(weight_two, hidden_one,
      output, s, assign);
  graphblas::backend::apply(output, output, sigmoid,
      assign);

  //Get elements from output vector
  int i[10]; int j[10]; double v[10];
  graphblas::backend::extracttuples(output, i,j,v);

  //Find highest scoring element and return it's
      class
  int result = 0;
  for (int i = 1; i < 10; i++) {
    if (v[i] > v[result]) result = i;
```

```
  }
  return i[result];
}

//Function for the training of an input. Updates the
    weights.
void Train(std::vector<ScalarType> element, int
    target) {
  //Needed for backend call
  graphblas::ArithmeticSemiring<ScalarType> s;
  graphblas::math::Assign<ScalarType> assign;
  graphblas::math::Accum<ScalarType> add;
  graphblas::math::Times<ScalarType> times;

  int result = Classify(element);

  //Initiate delta_out
  std::vector<std::vector<ScalarType>> v(10);
  int i[10]; int j[10]; double out[10]; graphblas::
      backend::extracttuples(output, i, j, out);
  double error = 0;
  for (int i = 0; i < 10; i++) {
    double a = -out[i];
    if (i == target) a = 1 - out[i];
    error += 0.5*a*a;
    double b = out[i] * (1 - out[i]);
    v[i] = { a*b };
  }
  delta_out = graphblas::CscMatrix<ScalarType>(v,
      0.0);

  //Transpose input, hidden_one and hidden_two to
      their corresponding vectors
  graphblas::backend::transpose(input, input_t);
      graphblas::backend::transpose(hidden_one,
      hidden_one_t);

  //Construct other delta's using matrix-matrix
      multiply and EwiseMult.
  graphblas::backend::transpose(weight_two,
      weight_two_t);
  graphblas::backend::mxm(weight_two_t, delta_out,
      delta_one, s, assign);
  graphblas::backend::apply(hidden_one, hidden_one,
      deriv, assign);
  graphblas::backend::ewisemult(delta_one, hidden_one
      , delta_one, times, assign);
  graphblas::backend::apply(delta_one, delta_one,
      learn_rate_enlarge, assign);
  graphblas::backend::apply(delta_out, delta_out,
```

```
        learn_rate_reduce, assign);

    //Updated the weights
    graphblas::backend::mxm(delta_one, input_t,
        weight_one, s, add);
    graphblas::backend::mxm(delta_out, hidden_one_t,
        weight_two, s, add);
  }

  //Test the performance of the network on images
      start_i till end_i
  void Test(std::vector<std::vector<ScalarType>> Img,
      std::vector<int> Label, int start_i, int end_i) {
    int correct[10] = { 0 }; int total[10] = { 0 }; int
        totalCorrect = 0; int totalElements = 0;
    for (int count = start_i; count < end_i; count++) {
      total[Label[count]]++;
      if (Classify(Img[count]) == Label[count]) correct
          [Label[count]] ++;
    }
    std::cout << "Scores: ";
    for (int n = 0; n < 10; n++) {
      totalCorrect += correct[n];
      totalElements += total[n];
      std::cout << n << ": " << correct[n] << "/" <<
          total[n] << ", ";
    }
    std::cout << "Total: "<< totalCorrect << "/" <<
        totalElements << std::endl;
  }

  //Execute a given amount of training reps on a batch
      of given size
  void RepTraining(std::vector<std::vector<ScalarType>>
      Img, std::vector<int> Label, int reps, int size)
      {
    std::cout << "Starting " << reps << " round(s) of
        training " << size << " images." << std::endl;
    for (int i = 0; i < reps; i++) {
      for (int n = 0; n < size; n++) {
        Train(Img[n], Label[n]);
      }
    }
  }
};

using namespace std;

//Required for Reading MNIST data
int ReverseInt(int i)
```

```
{
  unsigned char ch1, ch2, ch3, ch4;
  ch1 = i & 255;
  ch2 = (i >> 8) & 255;
  ch3 = (i >> 16) & 255;
  ch4 = (i >> 24) & 255;
  return((int)ch1 << 24) + ((int)ch2 << 16) + ((int)ch3
     << 8) + ch4;
}

void ReadMNIST(int NumberOfImages, int DataOfAnImage,
    vector<vector<ScalarType>> &arr)
{
  arr.resize(NumberOfImages, vector<ScalarType>(
     DataOfAnImage));
  ifstream file("C:\\t10k-images.idx3-ubyte", ios::binary
     );
  if (file.is_open())
  {
    int magic_number = 0;
    int number_of_images = 0;
    int n_rows = 0;
    int n_cols = 0;
    file.read((char*)&magic_number, sizeof(magic_number))
       ;
    magic_number = ReverseInt(magic_number);
    file.read((char*)&number_of_images, sizeof(
       number_of_images));
    number_of_images = ReverseInt(number_of_images);
    file.read((char*)&n_rows, sizeof(n_rows));
    n_rows = ReverseInt(n_rows);
    file.read((char*)&n_cols, sizeof(n_cols));
    n_cols = ReverseInt(n_cols);
    for (int i = 0; i<number_of_images; ++i)
    {
      for (int r = 0; r<n_rows; ++r)
      {
        for (int c = 0; c<n_cols; ++c)
        {
          unsigned char temp = 0;
          file.read((char*)&temp, sizeof(temp));
          arr[i][(n_rows*r) + c] = (ScalarType)temp /
             255;
        }
      }
    }
  }
}

vector<int> ReadTestLabels() {
```

```
  std::ifstream theFile;
  theFile.open("C:/testing_labels.txt");

  std::string line;
  std::vector<int> values;
  while (std::getline(theFile, line))
  {
    values.emplace_back(std::stoi(line));
  }
  return values;
}

//Main function
int main()
{
  //required
  srand(time(NULL));

  //Define size of the network
  graphblas::IndexType const Num_inputs = 784;
  graphblas::IndexType const Num_hidden_one = 30;
  graphblas::IndexType const Num_outputs = 10;
  graphblas::math::Assign<ScalarType> assign;

  //Create Network
  NeuralNet myNet;
  myNet.Fill(Num_inputs, Num_hidden_one, Num_outputs);

  //Filling test labels and images
  vector<int> testing_labels = ReadTestLabels();
  vector<vector<ScalarType>> testing_images;
  ReadMNIST(10000, 784, testing_images);
  std::cout << "Done parsing images!" << std::endl;

  //Train 6000 image 2 times
  myNet.RepTraining(testing_images, testing_labels, 2,
      6000);

  //Test performance on images 6000 to 7000
  myNet.Test(testing_images, testing_labels, 6000, 7000);

  //End
  return 0;
}
```